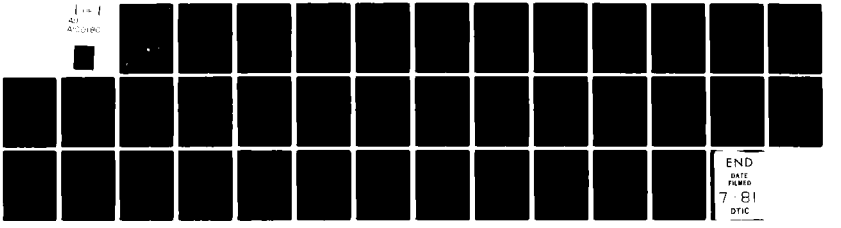
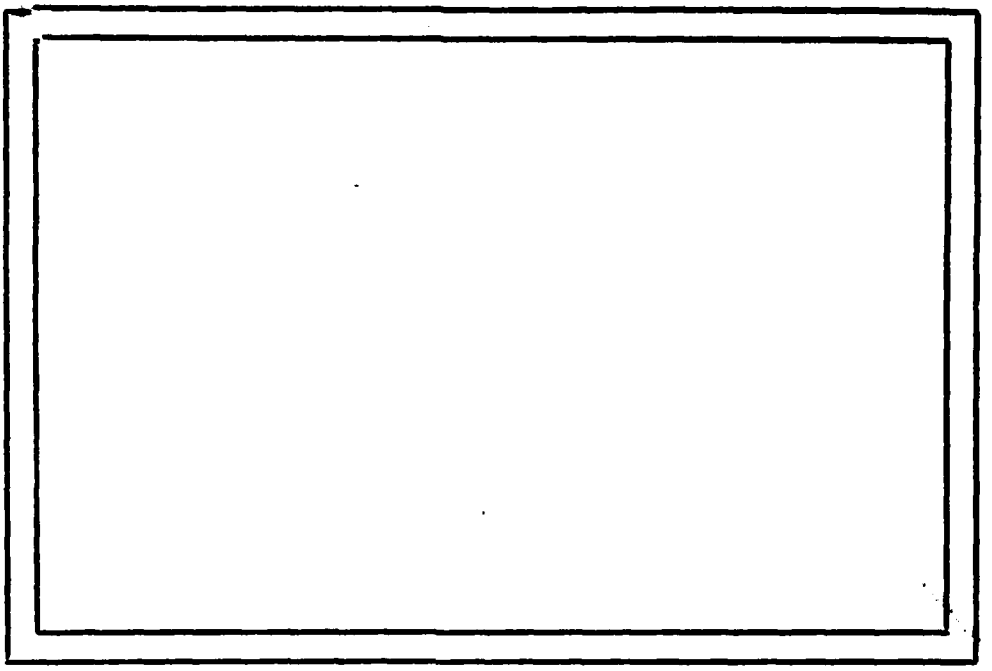
AD-A100 180 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
A FRAMEWORK FOR COMPUTER ARCHITECTURE.(U)
UNCLASSIFIED DEC 80 C A PERSY, L N DANNER, A K AGRAWALA AFOSR-78-3654
TR-974 AFOSR-TR-81-0486 NL



END
DATE FILMED
7-81
DTIC

LEVEL II

12



COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

Technical Report- 974 December 1980

A FRAMEWORK FOR COMPUTER ARCHITECTURE

By

Cornelia A. Persy
Lee N. Danner
Ashok K. Agrawala

This Research was supported in part by the Air Force Office of Scientific Research under grant AFOSR-78-3654 to the Department of Computer Science, University of Maryland.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-81-0486	2. GOVT ACCESSION NO.: AD-A100 180
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A FRAMEWORK FOR COMPUTER ARCHITECTURE.	5. TYPE OF REPORT & PERIOD COVERED: INTERIM
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Cornelia A. Persy, Lee N. Danner and Ashok K. Agrawala	8. CONTRACT OR GRANT NUMBER(s): AFOSR-78-3654
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science, University of Maryland, College Park MD 20740	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: PE 61102F, 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB DC 20332	12. REPORT DATE: DECEMBER 1980
	13. NUMBER OF PAGES: 32
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents a framework for computer architecture which is based on the principle function of a computer to perform a mapping from some input into an output. A set of recursive functions is developed to represent computer architecture at any desired level of detail. The definitions are insensitive to whether the functions are realized in software, hardware or firmware. The approach is illustrated using examples.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract

This paper presents a framework for computer architecture which is based on the principle function of a computer to perform a mapping from some input into an output. A set of recursive functions is developed to represent computer architecture at any desired level of detail. The definitions are insensitive to whether the functions are realized in software, hardware or firmware. The approach is illustrated using examples.

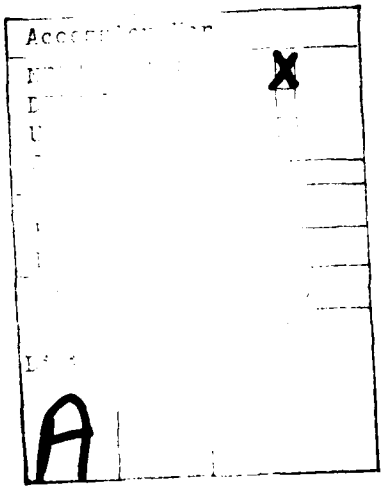


TABLE OF CONTENTS

1. Introduction

2. A general overview

3. Architecture structure

3.1 The algorithmic structure

3.1.1 The information structure

a. Objects

1.Atomic objects

2.Primitive objects

3.Complex objects

b. Operations upon objects

1.Arithmetic / logic operations

2.Relational operations

3.Structure operations

3.1.2 The control structure

a. Interpretation and transformation of objects

b. Execution order of operations upon objects

3.2 The realization

3.2.1 Resources

a. Processing units and processing elements

1.One processing unit systems

2.Arrays of identical processing elements

3.Pipelines of different processing elements

4.Multi-processing units systems

5.Distributed systems

b. Storage media

1.Registers

2.Random-access memories (RAM)

3.Content-addressable memories (CAM)

c. Channels

3.2.2 Rules

a. Cooperation between resources

b. Communication between resources

4. Formal definitions

4.1 Within the algorithmic structure

4.2 Within the realization

4.3 From the algorithmic structure into the realization

5. Examples

5.1 A matrix multiplication algorithm

5.2 A sorting algorithm

6. Concluding remarks

References

1. Introduction

The term computer architecture has been used since the early days of computers to imply the study of the structure of computer systems. These studies have focused on the organization and interconnection of the components of the system. The complexity of both the organization and the interconnection has been increasing steadily with developments in hardware technology. Software technology has also increased the complexity, size, and diversity of problems that use the facilities of computer systems. These combined growths have lead to a need for organizing the software and hardware parts of problem solving into a framework suitable for describing all users' interactions with the system. In this paper we present such a description whose resulting formal structure comprises a Framework for Computer Architecture. It is suitable for describing existing systems, usable in developing new systems, or in upgrading existing systems to take advantage of new technologies.

The emphasis in this framework is the problem solving aspect of the computer system, where a solution is developed from requirements using a process that consists of multiple translations or mappings. For example, to carry out a matrix inversion, the first step is to select an algorithm. A data structure for the problem has to be selected and a programming language chosen. The algorithm is then expressed in the programming language. The program is compiled to generate a relocatable code which is combined with necessary library routines to produce the absolute code. At the time of the

execution, the absolute code is loaded into the storage and the individual machine instructions are executed by invoking the appropriate sequences of microinstructions. Thus, the term computer architecture is used in this context to refer to the organization and interconnection of the hardware/software resources available at any level of this multiple level mapping process. Furthermore, the framework presented here provides a precise definition of computer architecture that delineates the various resources, the many functions they perform, and the controls necessary to enforce cooperation between them.

This definition of computer architecture could be applied to the analysis of distinct components of every specific architecture to allow optimization of each of these parts independently. Such an approach uses multiple recursive levels to represent this partitioning problem.

The theory is based on the principle function of a computer to perform a mapping from some input to an output. The defined functions (or mappings) are coextensive with the operations performed within the computer. Each operation allows recursive decomposition to represent functions of the subsystems, units, logical components within the units, etc. The abstract nature of the theory is insensitive to whether the functions are realized by hardware, software, or firmware. The recursive definitions allow one to decompose an operation to any desired level of detail, or to form compositions of detailed operations until one reaches an operation that embodies the entire architecture.

The given examples, a matrix multiplication and a bubble

sort, are explained in detail. Both examples were coded in Pascal and were executed on the Univac 1100/42 system.

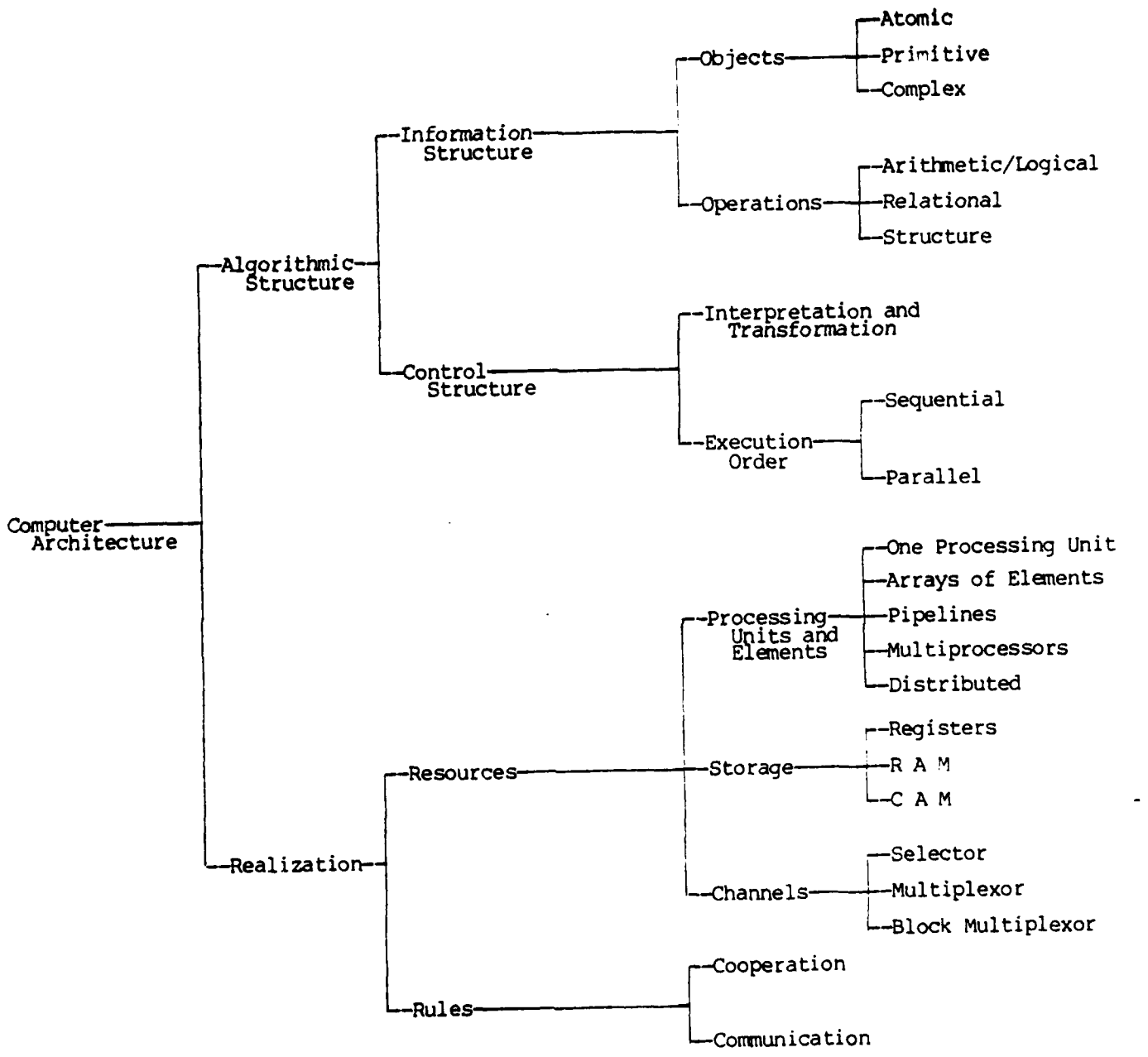
2. A General Overview

Computer architecture is characterized by the algorithmic structure and the realization. The algorithmic structure provides an organization for the data elements used to facilitate manipulation upon them. In general, a given problem is expressed in a coded form using objects (data types and data structures) and operations upon objects. Furthermore the interpretation and transformation of these objects and operations as well as the execution order of operations is contained in the algorithmic structure. The realization of a computer architecture is given by the type and number of the resources and a set of rules for communication and cooperation between them. In this paper we roughly distinguish between main hardware resources: processing units and elements, storage media, and channels. However, the emphasis lies on the cooperation rules between resources determining how the resources work together, and not on where the separations between the resources of the realization are chosen.

A set of recursive functions is developed to represent computer architecture to any desired level of detail.

3. Architecture Structure

The following figure depicts, graphically, the algorithmic structure and the realization with each partioned into several levels of detail.


Computer Architecture
Algorithmic Structure
Information Structure
Objects
Atomic
Primitive
Complex
Operations
Arithmetic/Logical
Relational
Structure
Control Structure
Interpretation and Transformation
Execution Order
Sequential
Parallel
Realization
Resources
Processing Units and Elements
One Processing Unit
Arrays of Elements
Pipelines
Multiprocessors
Distributed
Storage
Registers
R A M
C A M
Channels
Selector
Multiplexor
Block Multiplexor
Rules
Cooperation
Communication

3.1 The Algorithmic Structure

The algorithmic structure defines a set of abstract data types and data structures, which will be called objects, and the operations upon them. Furthermore it includes the specifications for the interpretation and transformation of objects and the execution order of operations upon them.

3.1.1 The Information Structure

The information structure of a digital computer architecture is composed of these objects and operations, while according to Shannon (1) information is a measure of one's freedom of choice when one selects a message, this term is more often used in the sense "the meaning associated with data" (2), and computers basically perform manipulations upon data elements. In order to facilitate these data manipulations, organizations for the data are defined as follows:

a. Objects

Within a program the valid data types and data structures are composed of the terminals of a language.

1. Atomic objects are the terminals of a language, like letters, digits, and special symbols.
2. Primitive objects (data types) are composed of atomic objects and a simple organization for their range of validity is given. For example, numbers, characters, and Booleans are primitive objects.
3. Complex objects (data structures) provide a

superordinated ranking for the primitive objects. Complex objects are composed of atomic objects and primitive objects with a more powerful organization. For example, arrays, stacks, queues, records, and files are complex objects.

b. Operations upon objects

Operations are the rules for data manipulation. An operation can be defined as the following mapping (or function):

m:{finite input set of data} -> {finite output set of data}

We basically distinguish between arithmetic/logic and relational operations upon primitive objects and structure operations upon complex objects.

1. Arithmetic/Logic operations

Operations are described by expressions, which consist of operators and functions applied to the primitive objects. Arithmetic operators are: + , - , * , / and the corresponding functions or mappings are SUM, DIFFERENCE, PRODUCT, and QUOTIENT. Logic operators are:$\wedge$,$\vee$,$\neg$, and the corresponding functions are AND, OR, and NOT.

2. Relational operations

They operate on primitive objects to produce a result that has the value true or false. Relational operators are: >,>=,<,<=,=,<>, and the corresponding functions are GREATER THAN, GREATER THAN OR EQUAL, LESS THAN, LESS THAN OR EQUAL, EQUAL, or NOT EQUAL.

3. Structure operations

Structure operations are performed on complex objects, like arrays, stacks, queues, records, and files. The describing expressions consist again of operators and functions applied to the complex objects. For example, typical stack operations are push and pop, which are composed of arithmetic expressions and the functions are: f1(stack) = push, f2(stack) = pop, respectively. Structure operations for records and files are typically open, close, read, write, etc. Selecting an array element by means of a subscript is also a structure operation.

3.1.2 The Control Structure

The control structure defines the interpretation and transformation of objects as well as the execution order of operations upon objects.

a. Interpretation and transformation of objects

The efficiency of a program is dependent upon the organization used for the objects to be processed. Therefore it is necessary to provide a suitable mechanism for the storing and processing of the objects. We can thus refer to the hardware elements which store the objects during program execution as the storage structure of a computer architecture. For example, in the von Neumann architectures it is often desirable to replace an object with the result

of an operation such that the result can be accessed as an operand for some further operation. In this case the variable mechanism (<name>,<value>) receives the new value using an assignment operator. Conversely, in the functional architecture (5) that deals with an expression by evaluating its subexpressions, the result is pushed back onto the top of the stack automatically, without requiring an assignment operation. The use of assignment, as part of the control structure for transforming objects, is further described in Example 5.1 .

b. Execution order of operations upon objects

The execution speed of a computer system is dependent on the speed of the technology and of the execution order of operations upon objects. Both sequential and simultaneous executions are considered.

Sequential. Von Neumann systems don't have any program structures or data structures which are recognizable by hardware. Consequently, in order to achieve parallelism, it has to be derived from the program structure and is normally done during the translation process using the language declarations that define the information and control structures of the program. The algorithmic structure of a von Neumann system is stamped by the von Neumann variable mechanism. A von Neumann variable is a pair (<name> , <value>), where <name> denotes the address of a storage cell and <value> specifies its contents. The value of a

variable can be changed arbitrarily often during program execution. A variable always has a defined value after it is once initiated. The control structure is strictly sequential.

The main hardware resources of a von Neumann system are: a CPU, a storage, and a connection for transferring words between the CPU and the storage. Backus called this connection the "von Neumann Bottleneck" (4). In order to avoid this bottleneck, Backus proposed to totally get rid of the von Neumann variable mechanism, which leads to functional or applicative programming (5). Variables or assignments don't exist and one deals only with expressions.

Giloi (3) takes the opposite approach and augments the variable mechanism by means of structured data sets. The augmented variable definition is the triple (<name> , <structure> , <value>). The special information unit containing the structure specification is called a variable descriptor. Variable descriptors may be:

- additional data types (like in tagged architectures)
- a level between instructions and data (like in pipeline machines).

Parallel Execution. One way of increasing the efficiency of a computer system, independent of faster technology, is by achieving a high degree of parallelism (3). Parallel executable activities could be:

- arithmetic operation (operation level)
- assignment statements (statement level)

- concurrent process (task level)
- user programs (job level)

The basic features of parallelism are dependent on the structure of the programs and objects. Implicit parallelism exists in many serial programs and can be recognized and exploited by investigating the control structure. For example, operations or sequences of operations that are executed repetitively and are independent in their order of execution are candidates for parallel execution (see Example 5.1).

For execution, it would be useful to alter the standard von Neumann variable mechanism by introducing eventual value variables, which can be temporarily undefined.

In explicit parallelism the program structures or the data structures are specified in a way that parallel execution is explicitly given and does not have to be derived from the program structure. We can distinguish several principles:

- parallel program structure (user specified within the program)
- independent data structure (for example, a file explicitly declared read only)
- ordering independent data sets in the form of an array
- self-describing information units
- self-identifying data.

Additionally there are the following combinations of sequential and parallel control flow in von Neumann systems:

- several sequential control flows at a time
- sequential control flow but concurrent execution of operations in each step
- sequential control flow with associative access to data.

3.2 The Realization

Giloi (3) claims that one important step in increasing the efficiency of a computer architecture can be achieved by a high degree of parallelism. In the algorithmic structure, definitions of abstract data types and operations upon them are required. Furthermore a program structure which grants explicit parallelism and variations from the strictly sequential control flow in von Neumann systems are needed.

Besides this, an important task lies in finding a close match between the control structure of an algorithmic structure and the cooperation rules for the resources of a computer architecture. The control structure determines the execution order of the operations upon objects, and the cooperation rules establish the principles according to which the resources are working together.

In general the realization of a computer architecture is given by the type and number of the main resources as well as the rules for communication and cooperation between them (described below).

3.2.1 Resources

a. Processing units and processing elements

A processing unit is an autonomous hardware resource which controls the program flow and executes the data transferring operations as well. In contrast to this, a processing element is only capable of executing the data transferring operations without controlling the program flow (examples are arithmetic/logical or floating point elements).

According to the number and arrangements of processing units and processing elements, one can distinguish the following systems:

1. One processing unit systems

   These are the conventional systems with one central processing unit (CPU). Special I/O devices are not taken into account. All von Neumann systems are of this type.

2. Arrays of identical processing elements

   The processing elements are arranged in the form of an array. This means, each element is only connected to its nearest neighbors. (The elements perform identical operations at a time.)

3. Pipelines of different processing elements

   In general, pipelines are a one dimensional arrangement of processing elements. But there are also pipelines of processing units. Normally the elements of a pipeline perform different operations at a time.

4. Multi-processing units systems

These systems consist of more than one processing unit. In the case that all processors have the same hardware but may take different functions within the system, they are called homogeneous otherwise they are called inhomogeneous. An asymmetric system is characterized by processing units performing different functions. Processing units within a symmetric system have interchangeable roles in the system.

5. Distributed systems

Distributed systems also consist of more than one processing unit; but in contrast to the multi-processing systems there is not a central supervisor, the control functions are distributed over the processor-storage-pairs of the system.

b. Storage media

A storage is a device used to retain information (data and programs) until it is used during execution. Note that in this paper the terms information and data are used loosely and not necessarily within the information theoretic context. According to their degree of complexity, storage media range from single registers to content-addressable memories (6).

1. Registers

A register is a simple device, consisting of a group of binary storage cells. Registers provide storage for units of information (bits, bytes, words, etc.). With

the addition of logic elements, registers can be transformed into counters, adders, shift-registers, and so on.

2. Random-access memories (RAM)

Random-access memories are a collection of registers that are addressable and have fixed or variable lengths. Every register is randomly accessible for reading or writing during any cycle. A subclass of the random-access memories are the read-only memories (ROM), storage devices that allow the read operation only. ROM's are usually faster and less expensive than standard RAM's.

3. Content-addressable memories (CAM)

A content-addressable memory allows access to a word by some portion of the content of the word rather than by its physical location. A key is specified as part of the input, and all words in memory that contain the key are available for reading or writing. All registers of the memory are accessed simultaneously and in parallel. CAM's are more expensive than RAM's because each cell must have storage capability as well as logic circuits for matching its content with an external argument.

c. Channels

Channels, also called I/O processors, provide a path for data-flow between I/O devices and storage media as well as the control-flow between I/O devices and processing units. Multiple devices may be connected to each channel. A

selector channel can service only one of its devices at a time. These channels are normally used for high-speed I/O devices. A multiplexor channel is able to simultaneously service many devices, but it accomplishes this only for slow I/O devices. A block multiplexor channel establishes a compromise between a selector channel and a multiplexor channel. It services only one device at a time but switches to perform an instruction for another device.

3.2.2 Rules

a. Cooperation between resources

A computer architecture consists of many resources, each capable of performing some part of a problem solution. In the algorithmic structure and the realization there is a potential for communication between parts of the information structure, the control structure, or the resources. Some are possible some are not; the possible interactions are delineated by the cooperation rules. For example, the development of an algorithm (information and control structure) assumes the existence of resources upon which the solution will be developed. The cooperation rules establish the configuration of the resources. They essentially determine how the resources can be used together and thus permit the communication among them. This is indicated in the algorithmic structure. For example, the operation i:=i+1 calls for an adder, and the fact that access to this device is possible is specified by the cooperation rules, whereas

the controls and physical transmission are done by the communication rules.

b. Communication between resources

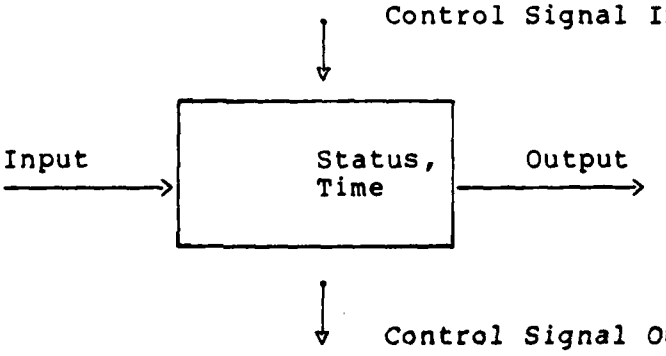
Communication between resources is controlled by protocols that manage the exchange of information between them. After a certain path among resources is defined by the cooperation rules, the protocols allow communication to be initiated, executed, and terminated. The actual communication is the transmission of information that includes both the data and the control signals.

As we have seen in the example given for cooperation rules a transmission is necessary to send the operands to the resource performing the operation (adder) and return the results. Control signals are also necessary to determine whether the adder is available (free) and to determine when the result is ready.

4. Formal Definitions

Formal definitions are presented describing the operations within the algorithmic structure and the realization, and a mapping from the algorithmic structure into the realization.

We like to view these operations as a function (or mapping) with the parameters input, output, control signal in, control signal out, status, and time, as shown in the following picture:



The objects are the input and output of the functions. The control signal in and the control signal out serve as operation identifiers. The status allows state changes to be passed from one operation to the next, like an arithmetic overflow passed from an arithmetic operation to a relational operation that follows. Thus the status contains exception conditions, machine status, and other similar information. Time describes the incremental time (delta t) that passes between the acceptance of the input and the completion of the output.

4.1 Formal Definition of Operations within the Algorithmic Structure

The output and control signal out of each operation can thus be defined as follows:

(1) output = f(input,control signal in, status, time)

(2) control signal out = f'(input, control signal in, status, time)

Furthermore, each individual operation might be broken up into a sequence of operations. The definitions (1) and (2) allow a recursive application in order to describe this sequence of operations:

(3) output = fn(parameters as above) o...o f0(parameters as above) , $n \in \mathbb{N}_0$

(4) control signal out = fn'(parameters as above) o...o f0'(parameters as above)

Each operation can thus be described as a composition of functions fn o...o f0 and fn' o...o f0'.

(5) operation : {f,f'} -> {fn o...o f0,fn' o...o f0'}

4.2 Formal Definition of Operations within the Realization

The operations within the communication rules are defined in the same way and lead to an equivalent definition to definition (5):

(6) operation : {g,g'} -> {gn o...o g0,gn' o...o g0'}

This definition allows again a recursive application to describe a sequence of operations.

4.3 Formal Definition of the Mappings from the Algorithmic Structure into the Realization

Operations within the algorithmic structure as well operations within the realization can be described by functions dealing with the same set of parameters. The functions are recursively applicable in order to describe each operation as a sequence of operations. Furthermore, it allows each operation within the algorithmic structure to be viewed as a sequence of operations within the realization. This is defined with the following mapping:

(7) M : {f,f'} -> {g,g'}

This definition grants that one can find a realization for a given program, but not every system may provide the resources in order to accomplish the realization for that program.

5. Examples

Two short examples are used to show the communication and cooperation rules as applied to the program statements of the realization of these problems.

5.1 A Matrix Multiplication Algorithm

The program for matrix multiplication (Figure 5.1) is written in the Pascal programming language and was executed on a UNIVAC 1100/42 system.

First, the algorithmic structure of the program will be described. The information structure is defined by the objects and the operations upon objects (described below).

The atomic objects are the keywords of the Pascal program, like: program, const, var, integer, begin, end, read, writeln, for, repeat, and until. Furthermore the special symbols, like: = , ; , : , etc. and the digits are atomic objects.

The primitive objects are the declared data types like constants and variables of the type integer. This example contains several primitive objects represented by the names imax, jmax, kmax, i, j, k. These names were chosen by the programmer.

The complex objects used in the program are array and file. The file structures are the special Pascal files input and output that are declared by Pascal as textfiles consisting of primitive objects that are characters. Array structures, defined by the programer are a, b, and c, and consist of a fixed number of primitive objects.

```
 1  program matmult(input,output);
 2  const imax = 4;
 3        jmax = 3;
 4        kmax = 2;
 5  var i,j,k: integer;
 6      a:array [1..imax,1..jmax] of integer;
 7      b:array [1..jmax,1..kmax] of integer;
 8      c:array [1..imax,1..kmax] of integer;
 9  begin
10  {
11       READ INPUT DATA FOR MATRICES A AND B
12  }
13  writeln (' Matrix A is ');
14  for i:= 1 to imax do begin
15      for j := 1 to jmax do begin
16          read (a[i,j]);
17          write (a[i,j]) end;
18      writeln end; {End of row}
19  writeln; {End of matrix A}
20  writeln ('Matrix B is ');
21  for j := 1 to jmax do begin
22      for k := 1 to kmax do begin
23          read (b[j,k]);
24          write (b[j,k]) end;
25      writeln end;
26  writeln;
27  {
28  COMPUTE PRODUCT MATRIX USING THREE LOOPING CONSTRUCTS
29  }
30  i := 1;
31  repeat {Beginning of row loop}
32      k := 1;
33      while k < kmax + 1 do begin {Column loop}
34          c[i,k] := 0;
35          for j := 1 to jmax do {Form element of C}
36              c[i,k] := c[i,k] + a[i,j] * b[j,k];
37          k := k + 1 end; {end of column loop}
38      i := i + 1
39  until i > imax; {End of row loop}
40  {
41       OUTPUT THE PRODUCT MATRIX B
42  }
43  writeln ('Matrix C = A x B is ');
44  for i := 1 to imax do begin
45      for k := 1 to kmax do
46          write (c[i,k]);
47      writeln end;
48  end.
```

Figure 5.1: Matrix Multiply Program Example.

The arithmetic/logic operations on objects used in this program are the arithemtic functions SUM and PRODUCT, expressed by the arithmetic operators + and *, respectively.

Relational operations are included within the while and until in lines 33 and 39. Structure operations are used to select elements from the array and file structures. The read and readln functions select elements from the structure input, and write and writeln functions place elements into the output structure. The Pascal selection operation for array structures is specified by following an array reference by a subscript, enclosed in square brackets. Thus the individual integer elements of the arrays a, b, and c, are selected throughout the program example.

In the control structure of the given Pascal program the execution order of operations upon objects is sequential. The program structure is determined by the compound statement begin - end, which indicates that a sequence of statements is to be executed in sequential order (7). However, there exist repetitive statements, like while, repeat, and for, which permit a program segment to be repeatedly executed as long as a specified condition is true.

Concurrent Pascal (9) allows multiple sequential programs to interact concurrently; but no explicit parallel operations are included in the language.

The transformation of the output from the operations in line 35 to be used in line 46 is through the assigment to

the complex object c. Similarly, results of line 37 are available to the relational operation in the while on line 33 through the use of assignment to the primitive object k.

This example (matrix multiplication) does contain an example of implicit parallelism in the following two lines:

```
35     for j := 1 to jmax do {FORM ELEMENT OF C}
36    c[i,k] := c[i,k] + a[i,j] * b[j,k];
```

Line 36 contains an assignment to array c[i,k] that is performed for subscripts of j ranging from one to jmax, and selecting elements from arrays a and b. Statement 36 also selects the element c[i,k] from array c; but is not dependent on any other elements of c. Consequently, all of the elements of the c array could be computed in parallel (*at the same time on multiple processing elements*); but would require the existence and use of explicit parallel operations.

Second, the realization of the program on the UNIVAC 1100/42 system is examined. This system is a multi-processing system and the main resources are: two processing units, a special I/O processing element, 16 channels, and five main storage media which are organized as random-access memories. The two processing units are homogeneous because they have the same hardware, but they perform different functions, therefore the system is asymmetric. Only one processing unit is connected to the special I/O processing element which has 16 channels attached to the I/O devices and secondary storage, like

disks and drums. Both processing units have access to the main storage media which are composed of three 131 K primary storages and the two 131 K extended storages.

The matrix multiply application program runs on this system using the following resources:
Only one processing unit can be used at a time by this program since its control structure implies sequential, rather than parallel, execution order and the Pascal compiler and the UNIVAC 1100 operating system used, do not recognize and support implied parallelism. Main storage and registers are used for containing the objects of the algorithmic structure. The channels are used in accessing the files named input and output, defined to the control structure by the Pascal compiler based on their reference in line 1.

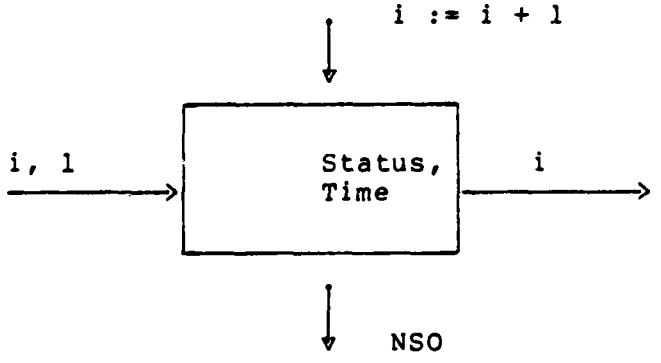
The cooperation rules provide the paths for the operations to be performed which deal with the parameters input, output, control signal in, control signal out, status, and time (defined formally in section 4.). In writing a program one takes for granted that the desired configuration of resources exists. A critical issue for the computer archtitect lies in finding a close match between the formal behavior of a program and the realization in a computer system.

The communication rules that have been derived from the control structure for Pascal programs are based on the language protocols that one normally takes for granted, that

the execution order is implicitly serial from the first line of the program down to the last line. Explicit changes are imposed by the occurance of a repeat-until, a begin-end with a for-do, a begin-end with a while-do, as in line 30 to 39, lines 21 to 25, and lines 33 to 37, respectivly.

A second level of implied communication rules is in line 36 where the operations + and * both appear. Pascal has an implied hierarchy of operations (protocols) that causes the function PRODUCT to occur before the function SUM . This hierarchy may be changed explicitly through the use of parenthesis. Other levels of communication rules force completion of the selection operation before accessing the data word from a complex object, and furthermore forces the arithmetic operations to await the arrival of the objects (from the RAM storage resouce to the register storage resource in the case of the UNIVAC 1100/42 system).

The communication rules, defined formally in section 4.2. deal with the parameters input, output, control signal in, control signal out, status, and time. For example the operation i := i+1 in line 38, would be described as follows:

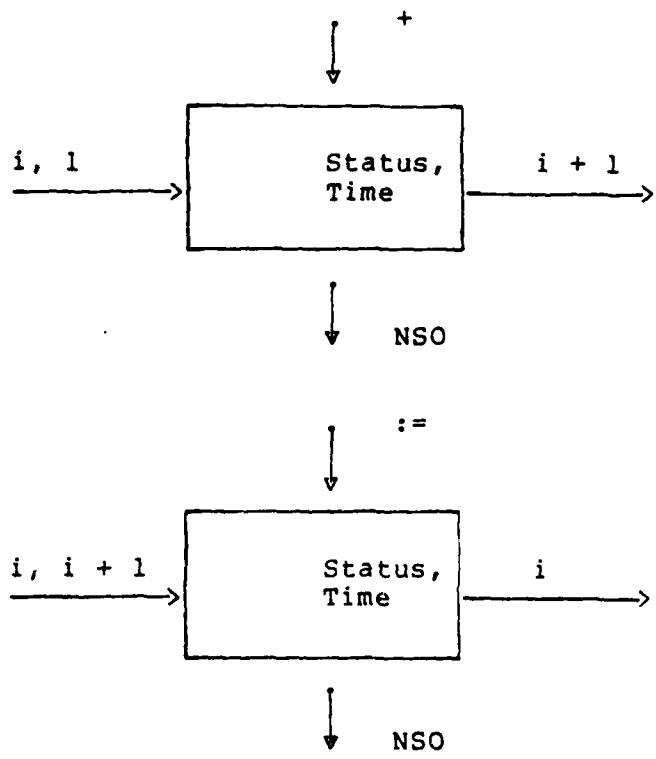


where the inputs are i and 1, the output is the new value of i, the control signal in is i := i+1, and the control signal out is next sequential operation (NSO). In the functional notation of equations (1) and (2):

$$i = g((i,1),\ i := i+1,\ \text{status},\ \text{time})$$

$$NSO = g'(\ \text{parameters as above}\ )$$

Furthermore, line 38 may be recursively decomposed into:



with the inputs and outputs and control signals in and out as shown. The functional notation of equations (3) and (4) express this as:

$$i = g_1((i,i+1),:=,s_1,t_1)\ \circ\ g_0((i,1),+,s_0,t_0))$$

$$NSO = g_1'(\text{param. as above})\ \circ\ g_0'(\text{param. as above})$$

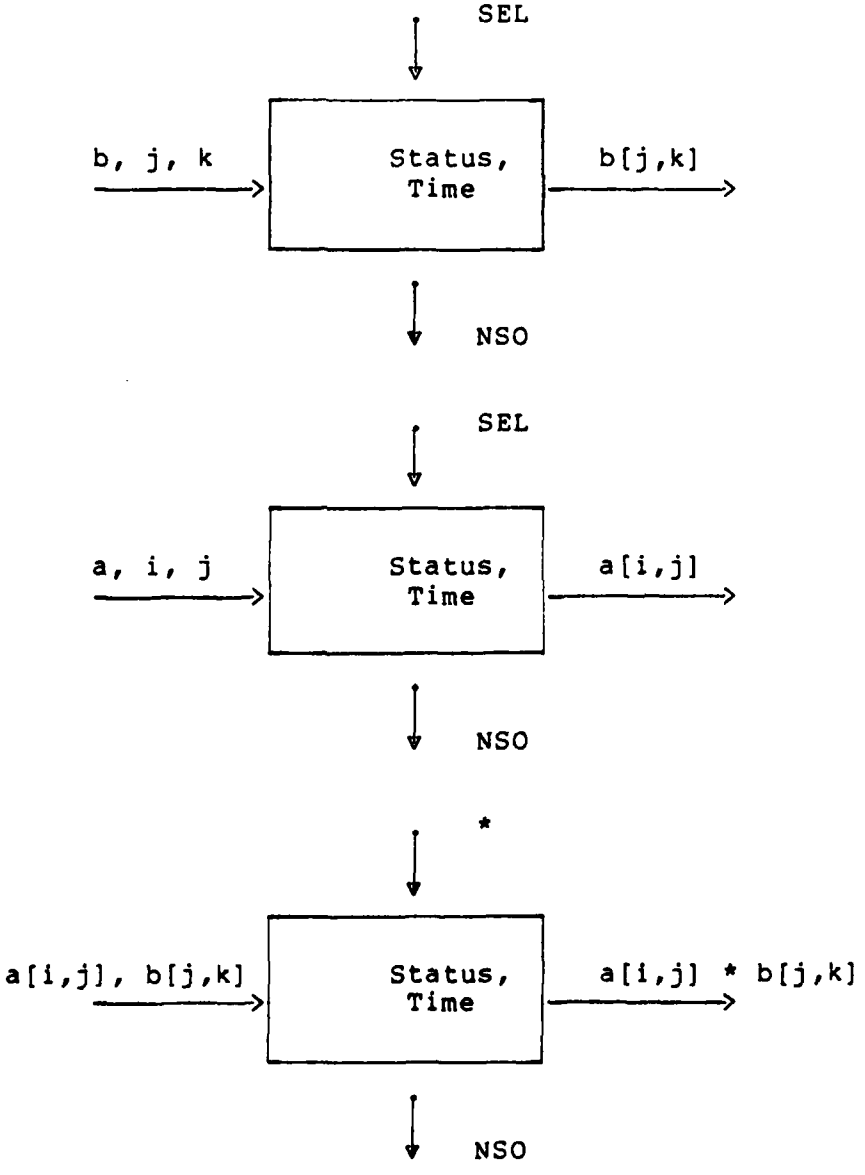
and the operation is:

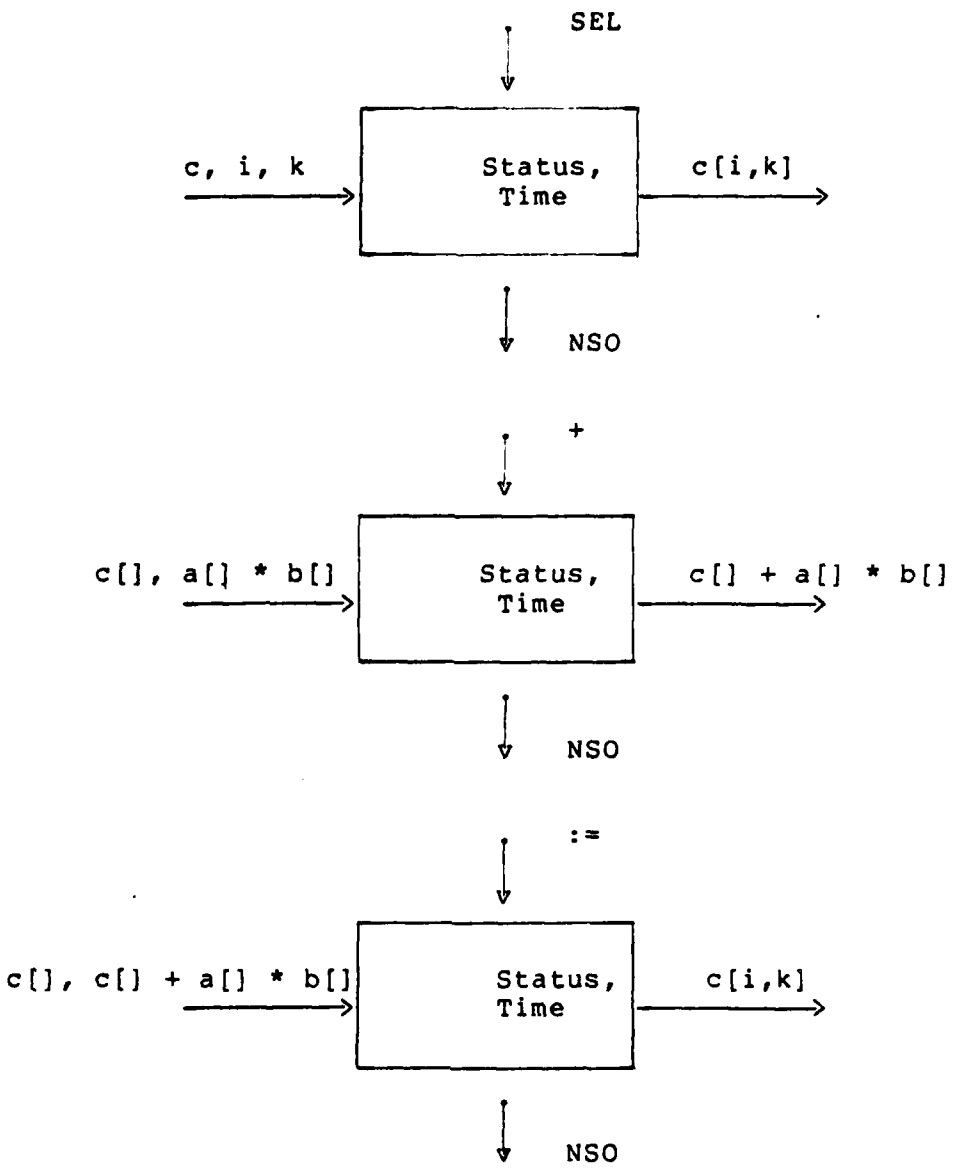
operation : {g,g'} -> {g og , g' og' }.

Similarly, the operation:

c[i,k] := c[i,k] + a[i,j] * b[j,k]

breaks up into the following operations, where SEL denotes the select operation and NSO stands for next sequential operation.





In viewing this theory, especially its recursive nature, one can see that operations within the algorithmic structure or within the realization might be composed of a sequence of operations. The given definitions (1-6) allow a recursively breaking up of each individual operation. Definition 7 provides a mapping from operations within the

control structure into the operations within the cooperation rules for the resources. This mapping assures that if the necessary resources exist a certain operation can be accomplished, otherwise the range of the mapping is empty.

With the opposite approach, it can be readily seen, an entire program or system of programs can also be represented as an operation.

5.2 A Bubble Sorting Algorithm (Figure 5.2)

Sorting differs significantly from matrix multiplication but the descriptions of the algorithmic structure and the realization of the two programs are substantially the same. One difference is that the sorting algorithm includes a relational operation as its principal operation. In line 22:

```
if data [n] < data [n-1] then begin
```

The relation between two adjacent elements selected from the array data is used to determine whether the elements are in sorted order. If they are not, the status condition from the relational operation causes the begin-end block of the then clause to be executed. If proper order is detected the begin-end block of the for in line 21 is advanced to the next value of n. The remainder of this example consists of an information structure and control structure similar to Example A.

The UNIVAC 1100/42 system was used for both examples.

```
 1   {
 2         SORT ROUTINE FOR DEMONSTRATION OF ARCHITECTURES
 3   }
 4   program dsort (input,output);
 5   const nmax = 10;
 6   var i,n,j,temp: integer;
 7       data: array [1..nmax] of integer;
 8   begin
 9   {
10        READ AND ECHO INPUT DATA
11   }
12   writeln (' UNSORTED DATA ARE ');
13   for i := 1 to nmax do begin
14       read (data[i]);
15       write (data[i]) end;
16   writeln;     {END OF DATA}
17   {
18        SORT LOOPS - OUTSIDE ( FOR LOOP) IS FORWARD SCAN
19                   - INSIDE (REPEAT-UNTIL LOOP) IS BKWD SCAN
20   }
21   for n := 2 to nmax do begin
22       if data [n] < data [n-1] then begin
23           j := n;
24           repeat
25               temp := data[j];
26               data [j] := data [j - 1];
27               data [ j - 1] := temp;
28               j := j - 1
29           until ((j = 1) or (data [j] > data [j - 1]))
30           end
31       end;
32   writeln (' SORTED DATA ARE:');
33   for i := 1 to nmax do begin
34       write (data [i]) end;
35   writeln
36   end.
```

Figure 5.2: Sort Program Example.

6. Concluding Remarks

This paper presented a framework for computer architecture which includes a methodology for formalizing the functions inherent in computer architeture. The artifical distinctions between hardware, software, or firmware common in the wide spectrum of the current state of the art in this field are eschewed. Detailed specifications of computer architecture have been aggregated into a number of basic constructs. The abstract nature of these basic definitions leads to a new view of the specifications in computer architecture. Many problems of previous descriptions are avoided by the functional decompositions provided by this approach. Similarly, the composition of individual parts is accomplished by recursively applying the basic operations. A primary benefit of this method is its extensability which allows support for new organizations of computer architectures utilizing recent technologies.

References

(1) Shannon, C. E. and W. Weaver, The mathematical theory of communication, University of Illinois Press, Urbana, Il, 1964

(2) Knuth, D. E., The art of computer programming, Vol. 1, Addison- Wesley, Reading, MA, 1968.

(3) Giloi, W. K., Rechnerarchitektur, Informatik - Spektrum 3, 3-18 (1980) Springer Verlag, 1980.

(4) Backus, J., "Can programming be liberated from the von Neumann style?", CACM, Vol. 21, No. 8, pp. 613 - 641.

(5) Kluge, W. E., The architecture of a reduction language machine hardware model, GMD, Bonn, Interner Bericht, 1979.

(6) Savage, J. E., The complexity of computing, John Wiley and Sons, Inc., 1976.

(7) Schneider, G. M., et. al., An introduction to programming and problem solving with Pascal, John Wiley and Sons, Inc., 1978.

(8) Madnick, S. E., and J. J. Donovan, Operating Systems, McGraw Hill, 1974

(9) Brinch Hansen, P., Concurrent Pascal, Transactions on Software Engineering, Vol. 1, No.2, pp. 199 - 207.